3. SYSTEM MANAGEMENT BUS

This laboratory work presents the System Management Bus (SMBus). After an overview of SMBus, bit and data transfers are described, the bus arbitration procedure is presented, the differences between SMBus and the I^2C bus are highlighted, and several command protocols are detailed. Next, the Intel SMBus controller is presented, including its registers and commands, as well as its use with I^2C devices. The applications aim to detect the devices connected to the computer's SMBus, read the contents of SPD memories present in the system, and decode the contents of these memories.

3.1. Overview of SMBus

System Management Bus (SMBus) is a simple serial bus with only two signal lines. This bus can be used for communication between various system devices and between these devices and the rest of a system. The operating principles of SMBus are similar to those of the I^2C bus. There are, however, several differences between the two buses, differences which will be presented later.

SMBus represents a control bus for system management and power management operations. A system may use the SMBus to transfer messages to and from various devices instead of using individual control lines, which allows to reduce pin count and interconnection wires. A device may use the SMBus to provide manufacturer information, provide the device model number, report different types of errors, accept control parameters, and return the device status.

SMBus was initially proposed by Intel as a link between an intelligent battery, a charger for the battery and a microcontroller that communicates with the rest of the system. The initial SMBus specification has been updated by the *System Management Interface Forum* (www.powersig.org). The current version of the SMBus specification is 3.2, released in 2022. Although the first versions of SMBus were primarily designed for smart batteries, the more recent versions allow to connect a wide variety of devices, including system sensors, memory chips, and communication devices.

Two types of devices, master devices and slave devices, may be connected to the SMBus. Each device connected to the bus is software addressable by a unique address. In general, a master device initiates a bus transfer between it and a single slave device, providing the clock signal required for the transfer. One exception occurs during initial bus setup, when a single master device may initiate transactions with multiple slave devices simultaneously. A slave device can receive data sent by the master device or it can provide data to the master device.

Like the I^2C bus, SMBus is a multi-master bus, allowing several master devices to be connected to the bus. However, only one device may control the bus at any time. Since several master devices may attempt to take control of the bus, SMBus provides an arbitration mechanism that relies on the wired-AND connection of all devices to the bus.

SMBus devices may be powered by the bus power source or by another power source (as in the case of smart batteries). Depending on the SMBus version, the nominal voltage of the bus power source, V_{DD}, is 5 V (minimum 4.5 V), 3 V (minimum 2.7 V), or 1.8 V (minimum 1.62 V). Figure 3.1 shows an example implementation of a 5-V SMBus with a device powered by the bus power source. At the same time, there is another device attached to the

SMBus lines, which is powered by a power source of 3 V. These devices will inter-operate as long as they adhere to the SMBus electrical specifications.

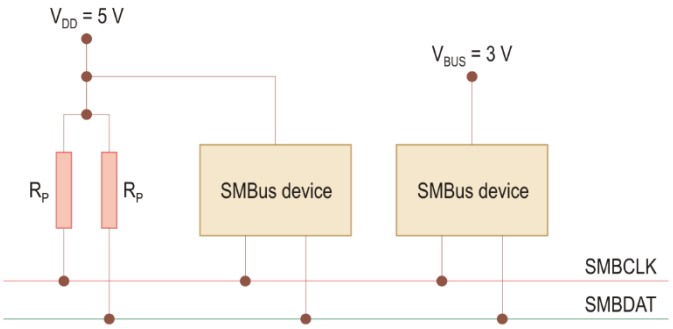

Figure 3.1. Example of SMBus topology.

The SMBCLK clock line and SMBDAT data line are both bidirectional, connected to the supply voltage through the R_P pull-up resistors; the lines can also be connected through a current source. The output stages of devices connected to the bus must have an open drain or open collector in order to perform the wired-AND function. When the bus is free, both lines are in the logic high state. A device may pull a bus line to the logic low state by driving the line to the defined low voltage level (maximum 0.4 V). When the device releases a bus line, it will be pulled to the logic high state by the R_P pull-up resistor.

Versions 1.0 and 1.1 of the SMBus specification, designed especially for smart batteries, defined only low-power electrical characteristics. These characteristics are appropriate for systems where conservation of energy is the most important aspect. Version 2.0 of the SMBus specification introduced a new class of higher-power electrical characteristics. These characteristics provide the robustness necessary, for example, to enable SMBus to traverse a PCIe connector, allowing SMBus devices on PCIe expansion cards to communicate with other SMBus devices on both the system board and other PCIe expansion cards. This version also reduced the nominal V_{DD} voltage from 5 V to 3 V. All these versions specify a clock frequency between 10 KHz and 100 KHz; most current implementations, however, use a clock frequency in the range of 50 KHz to 100 KHz.

Version 3.0 of the SMBus specification reduced the nominal V_{DD} voltage from 3 V to 1.8 V. This version also added new data protocols for reading or writing 32-bit and 64-bit values and introduced operation at the higher clock frequencies of 400 KHz and 1 MHz.

For devices that conform to the low-power electrical characteristics, the SMBus specification indicates a minimum current of 100 μA and a maximum current of 350 μA through the R_P pull-up resistor. Therefore, the minimum value of the R_P pull-up resistor should be 14.28 KΩ. For devices that conform to the high-power electrical characteristics, the specification indicates a minimum current sinking requirement of 4 mA, which determines a minimum value of the R_P pull-up resistor of 1.25 KΩ.

An optional feature of SMBus is the *Packet Error Checking* mechanism, which allows to improve reliability and communication robustness. This feature has been introduced in version 1.1 of the SMBus specification. It is implemented by appending a *Packet Error Code* (PEC) at the end of each message. The PEC is computed from all the bytes of a message by using an 8-bit cyclic redundancy check (CRC-8) code. The PEC byte is appended to the message by the device that supplied the last data byte.

3.2. Bit and Data Transfers

3.2.1. Bit Transfers

For the data to be valid, the SMBDAT line must be stable during the period in which the SMBCLK line is in the logic high state. The data line can only change state when the SMBCLK line is in the logic low state. This requirement is illustrated in Figure 3.2.

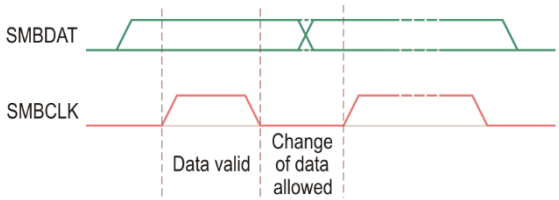

Figure 3.2. Data validity on the SMBus.

Each transfer begins with a START condition and finishes with a STOP condition. These conditions are always generated by the master device. START and STOP conditions are illustrated in Figure 3.3 and are defined below.

- A START condition is defined by a transition from the logic high state to the logic low state of the SMBDAT line while the SMBCLK line is in the logic high state.
- A STOP condition is defined by a transition from the logic low state to the logic high state of the SMBDAT line while the SMBCLK line is in the logic high state.

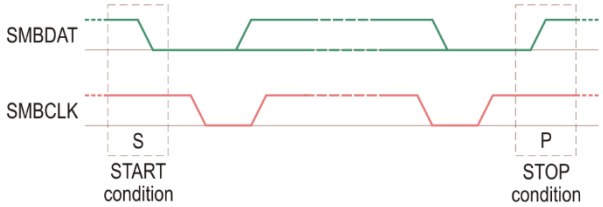

Figure 3.3. START and STOP conditions on the SMBus.

After a START condition, the bus is considered busy. The bus becomes idle again after a certain time following the STOP condition or after the SMBCLK and SMBDAT lines both remain in the logic high state for more than 50 μs.

3.2.2. Data Transfers

On the SMBus, bytes are transferred with the most significant bit (MSB) first. Each byte transferred on the bus must be followed by an acknowledge (ACK) bit. The clock pulse corresponding to the ACK bit is generated by the master device. The transmitter device, master or slave, releases the SMBDAT line during the ACK clock cycle. To acknowledge a byte, the receiver must pull the SMBDAT line to the logic low state during the ACK clock cycle. An example data transfer is illustrated in Figure 3.4.

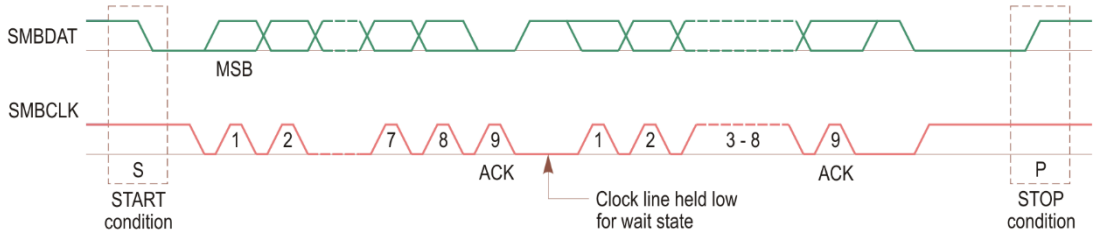

Figure 3.4. Example data transfer on the SMBus.

Note

- A device on the SMBus must always acknowledge its own address. This requirement is used by the SMBus controller to detect the presence of a detachable device on the bus.

It is possible for a receiver to not acknowledge a data byte and to send a NACK bit. For this, the receiver must maintain the SMBDAT line in the logic high state during the ACK clock cycle. A slave device may decide to not acknowledge a byte other than the address byte in the following situations:

- The slave device is busy performing a real-time task, or the requested data are not available. When the master device detects a NACK bit, it must generate a STOP condition to abort the transfer. As an alternative, the slave device can extend the logic low state of the clock signal (as illustrated in Figure 3.4) for a time of up to 25 ms to complete its tasks.
- The slave device detects an invalid command or invalid data. In this case, the master device must generate a STOP condition and retry the transaction.
- If a master-receiver device is involved in a transaction, it must signal to the slave-transmitter device not to send more data by transmitting a NACK bit after the last byte that was sent by the slave device. The slave-transmitter device must release the data line to allow the master device to generate a STOP condition.

Data transfers on the SMBus have the following format. After the START condition, the master device places on the bus the 7-bit address of the slave device it wants to address. The 7-bit address is followed by an eighth bit (R/W#) indicating the direction of the data transfer; if R/W# is zero, it indicates a write transfer, and if R/W# is one, it indicates a read transfer. A data transfer is terminated with a STOP condition generated by the master device.

Specific SMBus protocols require the master device to generate a repeated START condition followed by the slave device address without first generating a STOP condition.

3.3. Bus Arbitration

A master device may only start a transfer if the bus is in the idle state. One or more devices may generate a START condition at the same time. Since the devices that generated the START condition may not be aware that other master devices are contending for the bus, an arbitration procedure is used. Arbitration is performed with the SMBDAT line while the SMBCLK line is in the logical high state. A master device that transmits a logic high level on the SMBDAT line while other master devices are transmitting a logic low level on this line loses control of the bus in the arbitration cycle.

The master device that lost the arbitration may continue to provide clock pulses until transmission or reception of the byte on which it lost the arbitration is completed. If two master devices are arbitrating and the first master device wants to generate a repeated START condition, while the second master device wants to place a 0 data bit on the bus, the first master device will recognize that it cannot generate the START condition and will lose arbitration. If the first master device wants to generate a repeated START condition, while the second master device wants to place a 1 data bit on the bus, the second master device will detect the repeated START condition and will lose arbitration. If both master devices want to generate a repeated START condition in the same bit position, arbitration should continue at each data bit.

This arbitration mechanism requires that master devices participating in an arbitration cycle monitor the state of the SMBDAT line during arbitration. Once a master device has won arbitration, arbitration is disallowed until the bus will be again in the idle state.

3.4. Command Protocols

A typical SMBus device has a set of commands by which its data can be read or written. Each command is sent to a device using a specific command protocol that is defined by the SMBus specification. Version 3.0 of this specification defines a number of 15 command protocols. Devices need not support all the protocols defined in the SMBus specification.

Typical command protocols include an 8-bit command code. Command arguments and their return values may vary in length. Each command protocol (except for the *Quick Command* protocol) has two variants: one with the *Packet Error Code* (PEC) byte and one without the PEC byte.

The command protocols may have one of the following general formats:

- A master-transmitter device transmits data to a slave-receiver device. The transfer direction is not changed in this case.
- A master device reads data from a slave device immediately after the first byte. At the first acknowledge bit (provided by the slave-receiver device), the master-transmitter device becomes a master-receiver device (and the slave-receiver device becomes a slave-transmitter device).
- With the combined format, during a change of direction within a transfer, the master device generates a repeated START condition and sends the slave device address byte with the R/W# bit set to 1. The master-receiver device terminates the transfer by sending a NACK bit after the last byte of the transfer and then generating a STOP condition.

In this section, only a part of the SMBus command protocols is described. To simplify the presentation, the variants of command protocols with the PEC byte are not described. The following symbols are used to describe the command protocols:

- S: A START condition generated by a master device.
- Sr: A repeated START condition generated by a master device.
- P: A STOP condition generated by a master device.
- Wr: Indicates a write transfer; bit 0 (R/W#) of the address byte will be 0.
- Rd: Indicates a read transfer; bit 0 (R/W#) of the address byte will be 1.
- A: An ACK bit sent from a master device or a slave device.
- N: A NACK bit sent from a master device or a slave device.

Notes

- A number above a data field represents the length in bits of that field.
- The START and STOP conditions are transitions, not bits, and are shown without a bit count number above their symbols. The repeated START condition is also a transition rather than a bit.
- The fields containing information sent from a slave device to a master device are represented in a darker color.

3.4.1. Quick Command Protocol

In the *Quick Command* protocol (Figure 3.5), the R/W# bit of the slave device address represents the command. This bit may be used to enable/disable a device function or to enable/disable a low power operational mode. There are no data sent or received.

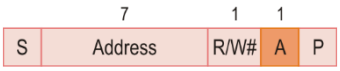

Figure 3.5. *Quick Command* protocol.

3.4.2. Send Byte Protocol

The *Send Byte* protocol (Figure 3.6) can be used to send a command encoded on a byte to a slave device. The command code byte follows the slave device address. All or parts of the byte sent may contribute to the command. For instance, the highest 7 bits of the byte might specify an action to be executed, such as placing the device in a specific operational mode, while the least significant bit may specify to enable or disable a device feature.

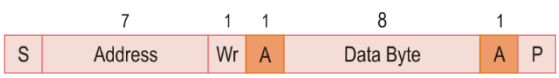

Figure 3.6. *Send Byte* protocol.

Note

- The interpretation of the command code is entirely device-specific. The SMBus specification does not define a list of command codes to be used with all SMBus devices.

3.4.3. Receive Byte Protocol

The *Receive Byte* protocol (Figure 3.7) is similar to the *Send Byte* protocol; the only difference is the direction of data transfer, which is from the slave device to the master device. This protocol can be used to read information from a slave device. A NACK bit indicates the end of the read transfer.

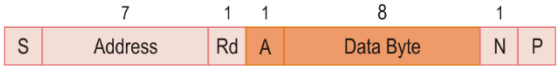

Figure 3.7. *Receive Byte* protocol.

3.4.4. Write Byte and Write Word Protocols

The *Write Byte* and *Write Word* protocols (Figure 3.8) enable to send one or two bytes of data to a slave device. The master device sends the slave device address followed by a write bit Wr. The slave device returns an ACK bit, and the master device sends the command code. The slave device again returns an ACK bit before the master device sends the data byte or word (with the low byte first). The slave device acknowledges each byte, and the entire transfer is finished with a STOP condition.

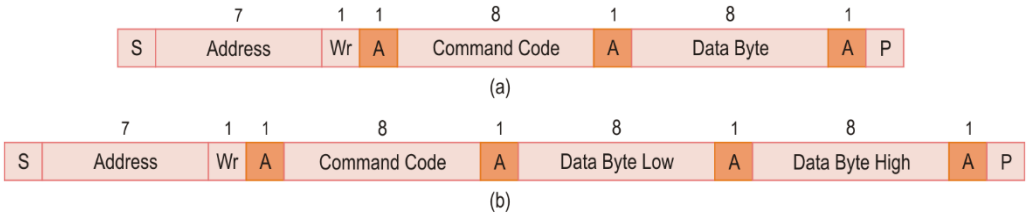

Figure 3.8. (a) *Write Byte* protocol; (b) *Write Word* protocol.

3.4.5. Read Byte and Read Word Protocols

The *Read Byte* and *Read Word* protocols (Figure 3.9) allow to read one or two bytes of data from a slave device. The master device must first send a command to the slave device. Then, it must follow the command with a repeated START condition to denote a read transfer from the addressed device. The slave device then returns one or two bytes of data. There is no STOP condition before the repeated START condition. The NACK bit indicates the end of the read transfer.

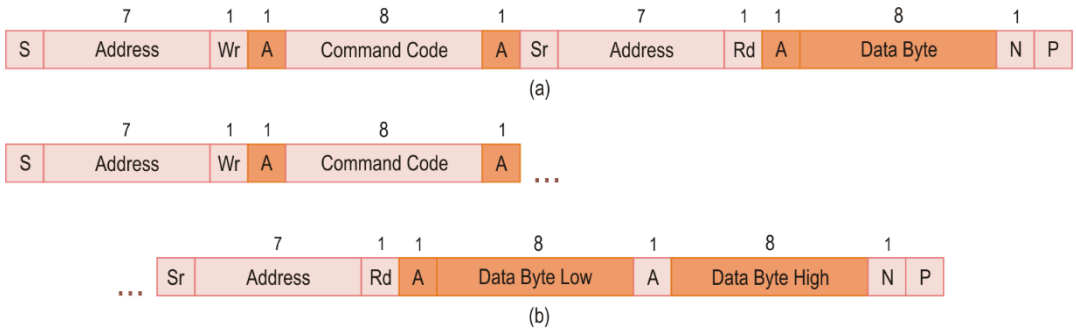

Figure 3.9. (a) *Read Byte* protocol; (b) *Read Word* protocol.

3.4.6. Process Call Protocol

In the *Process Call* protocol (Figure 3.10), the master device sends a command code followed by two bytes of data, and then waits for the slave device to return a two-byte value dependent of the data sent. This protocol is a combination of the *Write Word* protocol followed by the *Read Word* protocol without the *Read Word* command code and the *Write Word* STOP condition. There is no STOP condition before the repeated START condition. The NACK bit indicates the end of the transfer.

Figure 3.10. *Process Call* protocol.

3.4.7. Block Write and Block Read Protocols

In the *Block Write* protocol (Figure 3.11 (*a*)), the master device first sends a slave device address and a write bit, followed by a command code. Then, the master device sends a byte count which indicates the number of data bytes that will follow in the message. The byte count may be zero. This protocol allows to transfer a maximum of 255 data bytes.

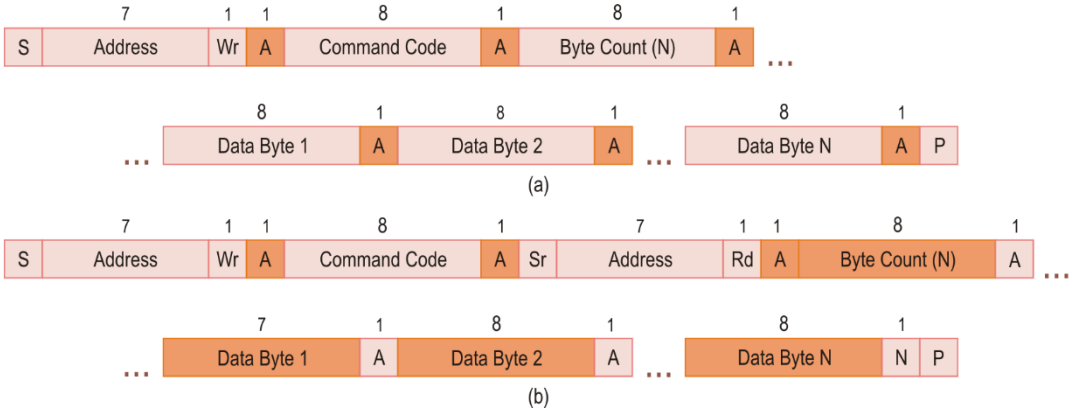

Figure 3.11. (a) *Block Write* protocol; (b) *Block Read* protocol.

The *Block Read* protocol (Figure 3.11 (*b*)) differs from the *Block Write* protocol in that a repeated START condition is inserted in order to allow the change of the transfer direction. A NACK bit immediately preceding the STOP condition indicates the end of the read transfer.

3.4.8. Block Write-Block Read Process Call Protocol

The *Block Write-Block Read Process Call* protocol (Figure 3.12) has two parts. In the first part of the protocol, the master device sends the slave device address, a write bit, the command code, and a write byte count (M) that specifies how many more bytes will be sent in the first part of the protocol. The write byte count may be zero. In the second part of the protocol, the master device generates a repeated START condition, then sends the slave device address and a read bit. The slave device will send a read byte count (N), followed by N data bytes. The read byte count may differ from the write byte count and may be zero. The combined byte count (M + N) must not exceed 255. Note that there is no STOP condition before the repeated START condition.

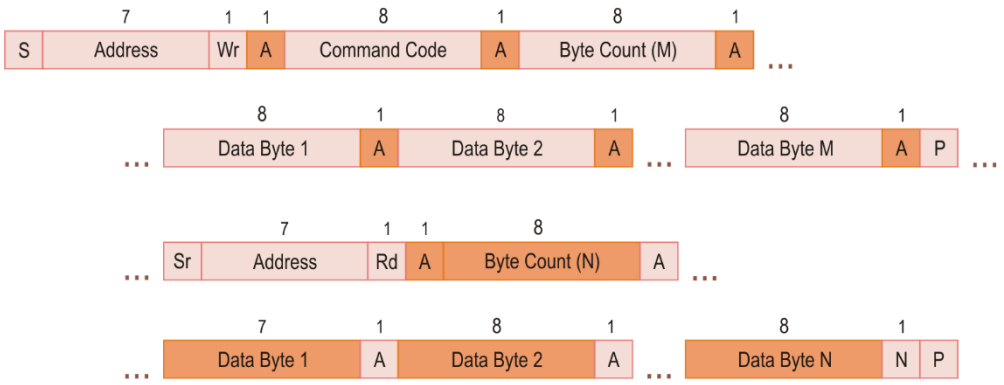

Figure 3.12. *Block Write-Block Read Process Call* protocol.

3.5. Differences between SMBus and I²C Bus

SMBus is derived from the I²C bus, and therefore the two buses are very similar. However, there are several differences between these buses regarding electrical characteristics, timing, operation modes, and protocols. The most important differences are described next.

- The minimum and maximum values of the V_{DD} supply voltage are specified differently for the SMBus and I²C buses. The SMBus specification restricts the nominal supply voltages of devices attached to the bus to a minimum of 1.8 V and a maximum of 5 V. The I²C specification is more tolerant regarding the values of the supply voltage.
- The I²C and SMBus specifications define the low-level and high-level input voltages for the bus lines differently. The I²C specification defines the low-level input voltage (V_{IL}) as 30% of V_{DD}, and the high-level input voltage (V_{IH}) as 70% of V_{DD}. The SMBus specification defines fixed thresholds for these voltages; V_{IL} is defined as maximum 0.8 V, and V_{IH} as minimum 1.35 V. However, even with the different specifications for the input voltage thresholds, in general, I²C and SMBus devices will be interoperable over the supply voltages allowed by the SMBus specification.
- The SMBus specification limits to 10 ms the maximum amount of time a master device may extend the logic low state of the clock signal within each byte of a message. There is also a limit of 25 ms on the total time a slave device may extend the logic low state of the clock signal within each message. A further restriction on the SMBus operation is a timeout interval of 35 ms, after which the bus is presumed hung and all devices attached to the bus must reset their I/O interfaces. In contrast to the SMBus specification, the I²C specification allows a master or a slave device to hold the clock line in the logic low state for an unlimited amount of time, and there is no timeout interval defined.
- The SMBus specification defines a bus with low-power electrical characteristics for applications where power consumption must be minimized, such as in battery-powered systems. The I²C bus does not have a similar specification.
- The SMBus specification requires a minimum operating frequency of 10 KHz, while the I²C specification does not indicate a minimum bus frequency.
- The I²C specification does not require that a slave device always acknowledge its own address. Consequently, if a device does not acknowledge its address, the controller cannot detect whether the device is busy, has failed, or it has been removed from the bus. To eliminate this confusion, the SMBus specification requires that a slave device always acknowledge its own address.

- The I^2C specification only defines three types of protocols or bus cycles: write cycle, read cycle, and combined format cycle (these cycles are described in Section 3.7.1). The SMBus specification defines a larger number of command protocols that may be used for communication with devices. Therefore, not all I^2C devices will support the SMBus command protocols.
- An important difference between SMBus and the I^2C bus is how the number of bytes transferred is controlled during block write and block read transfers. During I^2C write cycles, the slave device is the one that determines the size of the data transfer by sending a NACK bit after the last data byte accepted. During the SMBus *Block Write* protocol, however, the master device determines the transfer size by sending the byte count as part of the protocol; after the last byte sent, the master device expects a normal acknowledgement (ACK bit) from the slave device. During I^2C read cycles, the master device is the one that controls the transfer size. After it receives the last data byte, the master device sends a NACK bit and generates a STOP condition, ending the cycle. During the SMBus *Block Read* protocol, however, the slave device is the one that determines the transfer size. After it returns the last data byte, the slave device sends a NACK bit, and then the master device generates a STOP condition, ending the transfer.
- An I^2C slave device sends a NACK bit to indicate that it cannot receive any more data bytes. A SMBus slave device sends a NACK bit to indicate the reception of an invalid command or data.
- During several SMBus protocols, a command code byte is sent after the slave device address. I^2C slave devices interpret this byte as the first write data byte in the data block.
- The SMBus protocols for reading data from a device generally use a repeated START condition. Some I^2C devices may not recognize a repeated START condition. An attempt to use a SMBus protocol with this condition to read data from an I^2C device may yield unexpected results.

3.6. The Intel SMBus Controller

The *Platform Controller Hub* (PCH) component of current Intel chipsets includes a SMBus controller, which represents the PCIe device 31, function 3 (D31:F3) or device 31, function 4 (D31:F4). The features of this controller depend on the chipset series. For instance, the SMBus controller of the Intel 8 Series chipset used in the laboratory computers allows a frequency of operation up to 100 KHz (with transfer speeds of up to 100 Kbits/s) and supports the SMBus specification version 2.0. This controller also supports communication with many devices that are compatible with the I^2C bus.

3.6.1. SMBus Controller Registers

The Intel SMBus controller contains two categories of registers: PCI configuration registers and I/O registers. The following sections describe the most important registers from each category.

3.6.1.1. PCI Configuration Registers

The PCI configuration registers of the Intel SMBus controller include the PCI configuration header registers and a single PCI device-specific register. These registers can be accessed using either the PCI-compatible configuration mechanism or the PCIe enhanced configuration mechanism. The PCI configuration registers are listed in Table 3.1. Each entry of the table contains the offset of a PCI configuration register, its mnemonic, name, and size. The offset is relative to the base address of the configuration space allocated for function 3 or function 4 of PCIe device 31.

Table 3.1. Intel SMBus controller PCI configuration registers.

Offset	Mnemonic	Register Name	Size (Bits)
0x00	VID	Vendor Identification	16
0x02	DID	Device Identification	16
0x04	PCICMD	PCI Command	16
0x06	PCISTS	PCI Status	16
0x08	RID	Revision Identification	8
0x09	PI	Programming Interface	8
0x0A	SCC	Sub-Class Code	8
0x0B	BCC	Base Class Code	8
0x10	SMBMBAR0	Memory Base Address Register 0	32
0x14	SMBMBAR1	Memory Base Address Register 1	32
0x20	SMB_BASE	SMBus Base Address	32
0x2C	SVID	Subsystem Vendor Identification	16
0x2E	SID	Subsystem Identification	16
0x3C	INT_LN	Interrupt Line	8
0x3D	INT_PN	Interrupt Pin	8
0x40	HOSTC	Host Configuration	8

Note

- In the SMBus-e.h header file, which will be used for the applications, the SMBus controller's PCI configuration registers are defined in a structure named `SMBUS_CFG`.

PCI Configuration Header Registers

The SMBus controller's PCI configuration header registers (with offsets between 0x00 and 0x3D) have the same functions as the general PCI configuration header registers described in the laboratory document *PCI Express Bus*. The SMBMBAR0 register contains on bit positions 31 through 8 the low part of the base address for the memory mapped I/O registers, while the SMBMBAR1 register contains on bit positions 31 through 0 the high part of the base address for the memory mapped I/O registers. The SMB_BASE register contains on bit positions 15 through 5 the base address for the I/O registers mapped in the I/O space.

HOSTC – Host Configuration Register

The HOSTC register is specific to the SMBus controller. The bits of this register are described next.

- Bits 7..5 are reserved.
- Bit 4 (SPD Write Disable): If set, writes to SMBus addresses between 0x50..0x57 are disabled. This address range is used by SPD (*Serial Presence Detect*) memories; these are serial-access EEPROM memories used in DIMM memory modules.
- Bit 3 (Soft SMBus Reset): If set, the SMBus state machine and logic is reset.
- Bit 2 (I2C_EN): If set, some command protocols of the SMBus controller are changed in order to be able to communicate with I^2C devices. The effects of setting this bit are discussed in Section 3.6.2.5 and Section 3.6.2.6.
- Bit 1 (SMB_SMI_EN): If set, any source of a SMBus interrupt will be routed to generate a special *System Management Interrupt* (SMI).
- Bit 0 (HST_EN): If set, the SMB controller is enabled to execute commands.

3.6.1.2. I/O Registers

The most important I/O registers of the SMBus controller are listed in Table 3.2.

Table 3.2. Intel SMBus controller I/O registers.

Offset	Mnemonic	Register Name	Size (Bits)
0x00	HST_STS	Host Status	8
0x02	HST_CNT	Host Control	8
0x03	HST_CMD	Host Command	8
0x04	XMIT_SLVA	Transmit Slave Address	8
0x05	HST_D0	Host Data 0	8
0x06	HST_D1	Host Data 1	8
0x07	Host_BLOCK_dB	Host Block Data Byte	8
0x08	PEC	Packet Error Check	8
0x0C	AUX_STS	Auxiliary Status	8
0x0D	AUX_CTL	Auxiliary Control	8

Notes

- The SMBus controller's I/O registers can be accessed either as memory mapped registers using the contents of SMBMBAR0 and SMBMBAR1 registers as base address, or registers mapped in the I/O space using the contents of SMB_BASE register as base address. The offsets are the same for both memory mapped registers and registers mapped in the I/O space.
- In the SMBus-e.h header file, the SMBus controller's memory mapped I/O registers are defined in a structure called `SMBUS_REG`.

HST_STS – Host Status Register

The HST_STS register contains status bits set by the SMBus controller. A status bit can be cleared by the software writing 1 to the particular bit position. Writing 0 to any bit position has no effect. The bits of this register are described next.

- Bit 7 (BYTE_DONE_STS): If set, the controller received a byte (for a block read command) or it has completed transmission of a byte (for a block write command) when the 32-byte buffer is not enabled. This bit has no meaning for block transfers when the 32-byte buffer is enabled.
- Bit 6 (INUSE_STS): If set, the SMBus controller is in use by a software thread. This bit can be used as semaphore among various software threads that may need to use the SMBus controller. Software may check the status of this bit until it reads 0, and then it may use the controller. Subsequent reads of this bit will return 1. A write of 1 to this bit will reset the next read value to 0.
- Bit 5 (SMBALERT_STS): If set, the source of an interrupt was the *SMBALERT#* signal. This is an optional interrupt signal of SMBus which can be used by a slave device to signal the controller that it has a message to send. The use of this signal is not described in this laboratory work.
- Bit 4 (FAILED): If set, the source of an interrupt was a failed bus transaction. This bit is set in response to the KILL bit of the *Host Control* register being set in order to terminate a transaction.
- Bit 3 (BUS_ERR): If set, the source of an interrupt was a collision during execution of a transaction.
- Bit 2 (DEV_ERR): If set, the source of an interrupt was either an invalid command field or a timeout error generated by the controller.
- Bit 1 (INTR): If set, the source of an interrupt was the successful completion of the last command. The INTR bit is not dependent on the status of the INTREN bit of the *Host Control* register. If the INTREN bit is not set, then the INTR bit will be set at the successful termination of a command, although the interrupt will not be generated.

- Bit 0 (HOST_BUSY): If set, it indicates that the SMBus controller is running a command. No registers of the controller should be accessed while this bit is set, except for the *Host Block Data Byte* register. This bit can be used to detect when the controller has finished execution of a command.

HST_CNT – Host Control Register

The HST_CNT register allows specifying the command to be executed by the SMBus controller, initiate execution of the specified command, and to enable the generation of an interrupt when a command has been completed. The bits of this register are described next.

- Bit 7 (PEC_EN): If set, the SMBus controller will perform the bus transaction with the *Packet Error Checking* (PEC) phase appended. For write transactions, the value of the PEC byte is transferred from the PEC register. For read transactions, the PEC byte is loaded into the PEC register. This bit must be set prior to the write operation in which the START bit is set.
- Bit 6 (START): When this bit is set, the controller will start execution of the command specified in the SMB_CMD field of the *Host Control* register. All registers needed for the command execution should be set prior to setting this bit.
- Bit 5 (LAST_BYTE): This bit is used for the *Block Read* and *I^2C Read* commands. Software sets this bit to indicate that the next byte will be the last byte to be received in the data block. This causes the controller to send a NACK bit (instead of an ACK bit) after receiving the last byte.
- Bits 4..2 (SMB_CMD): Software writes into this field the command code to be executed by the controller. The encodings of supported commands and their names are listed below; the commands are described in Section 3.6.2.

  000: *Quick Command*;
  001: *Send/Receive Byte*;
  010: *Write/Read Byte*;
  011: *Write/Read Word*;
  100: *Process Call*;
  101: *Block Write/Read*;
  110: *I^2C Read*;
  111: *Block Write-Block Read Process Call*.

- Bit 1 (KILL): If set, the controller aborts the current transaction, sets the FAILED status bit, and generates an interrupt if the interrupts are enabled. Once set, this bit must be cleared by the software to allow the controller to operate normally.
- Bit 0 (INTREN): If set, this bit enables the generation of an interrupt when execution of a command is completed.

HST_CMD – Host Command Register

The contents of the *Host Command* register are transmitted by the controller in the *Command Code* field of the SMBus protocols during the execution of several commands.

XMIT_SLVA – Transmit Slave Address Register

The *Transmit Slave Address* register is written by the software with the slave device address and direction bit. The contents of this register are transmitted by the controller in the *Address* field of any SMBus protocol.

- Bits 7..1 (Address): This field contains the 7-bit address of the slave device.

- Bit 0 (RW): This bit indicates the direction of the transfer. A value of 0 indicates a write transfer, while a value of 1 indicates a read transfer.

HST_D0 – Host Data 0 Register

For the *Write Byte* command, the *Host Data 0* register is written by the software with the data byte to be sent in the *Data Byte* field of the SMBus protocol. For the *Write Word* command and *Process Call* command, this register is written by the software with the data byte to be sent in the *Data Byte Low* field of the SMBus protocol. For the *Block Write* and *Block Write-Block Read Process Call* commands, this register is written by the software with the number of bytes to be transferred, number which will be sent in the *Byte Count* field of the SMBus protocol.

Note

- For the *Block Write* and *Block Write-Block Read Process Call* commands, the *Host Data 0* register should be written with a value between 1 and 32 for the byte count. A value of 0 or above 32 will result in unpredictable behavior since the controller does not check the validity of the byte count.

With the *Read Byte* command, after the command completes, the *Host Data 0* register will contain the data byte read. With the *Read Word* and *Process Call* commands, this register will contain the data byte received in the *Data Byte Low* field of the SMBus protocol. With the *Block Read* and *Block Write-Block Read Process Call* commands, this register will contain the number of bytes to be received, number which is received in the *Byte Count* field of the SMBus protocol.

Note

- With the *I^2C Read* command, each data byte read is stored in the *Host Block Data Byte* register and not in the *Host Data 0* register.

HST_D1 – Host Data 1 Register

For the *Write Word* and *Process Call* commands, this register is written by the software with the data byte to be sent in the *Data Byte High* field of the SMBus protocol. For the *Read Word* and *Process Call* commands, this register will contain the data byte received in the *Data Byte High* field of the SMBus protocol. Other commands do not use this register.

Host_BLOCK_dB – Host Block Data Byte Register

Host Block Data Byte is either a register or a pointer into a 32-byte buffer, depending on whether the E32B bit has been set in the *Auxiliary Control* register. When the E32B bit has been cleared, this is a register containing a data byte to be sent with a block write transfer or received with a block read transfer. When the E32B has been set, this register is used as a pointer to access the 32-byte buffer. This pointer is reset to 0 by reading the *Host Control* register. The pointer then increments automatically upon each access to this register.

Details about using the *Host Block Data Byte* register are presented in Section 3.6.2.6.

PEC – Packet Error Check Register

For a write command, the *Packet Error Check* register is written with the 8-bit CRC code representing the PEC byte prior to starting the command. For a read command, the PEC byte is loaded into this register from the bus and then it can be read by software.

AUX_STS – Auxiliary Status Register

The main function of the *Auxiliary Status* register is to signal a CRC error in a received message.

- Bit 0 (CRC Error – CRCE): If set, it indicates that the received message contains a CRC error. When this bit is set, the DEV_ERR bit of the *Host Status* register will also be set.

AUX_CTL – Auxiliary Control Register

The *Auxiliary Control* register allows to enable or disable the 32-byte buffer and to enable or disable appending the CRC code to a message.

- Bit 1 (Enable 32-Byte Buffer – E32B): If set, the *Host Block Data Byte* register is a pointer into the 32-byte buffer, as opposed to a single data register. This enables the block commands to send or receive up to 32 data bytes before the controller generates an interrupt.
- Bit 0 (Automatically Append CRC – AAC): If set, the controller will automatically append the CRC code to the message. This bit must not be changed during bus transactions.

3.6.2. SMBus Controller Commands

The commands supported by the SMBus controller implement the SMBus protocols that have been described in Section 3.4. In addition, the controller supports the *I^2C Read* command, which allows to read data from I^2C-compatible devices that do not implement the SMBus block read protocols.

For issuing a certain command, the software should perform the following operations:

1. Wait until the SMBus controller finishes a previous command, when it clears the HOST_BUSY bit in the *Host Status* register.
2. Set the controller's registers needed for the command; the registers that should be set for each command are presented in subsections 3.6.2.1 through 3.6.2.8.
3. Write to the *Host Control* register a byte containing the command code in the SMB_CMD field and the START bit set to 1. Assuming that no interrupts are used, bit 0 (INTREN) of this byte should be 0.

After issuing a command, the *Host Status* register is used to determine the progress of the command. If the command completes successfully, the INTR bit will be set. If the device does not respond with an ACK bit and the transaction times out, the DEV_ERR bit will be set. If the software sets the KILL bit in the *Host Control* register during the command execution, the transaction will be aborted, and the FAILED bit will be set in the *Host Status* register.

Assuming that no interrupts are used, the software should perform the following operations to determine if the command has completed and to check the completion status:

1. Wait the completion of command execution, when the controller sets either the INTR bit in the *Host Status* register, or one of the following bits in the same register: FAILED, BUS_ERR, or DEV_ERR.
2. After command completion, if the INTR bit is set, the command has completed successfully. If one of the FAILED, BUS_ERR, or DEV_ERR bit is set, the command completed with an error and the software may signal the type of error. In either case, the software should clear the status bits in the *Host Status* register by reading the contents of the register and writing back the value read.

3.6.2.1. Quick Command

Before issuing a *Quick Command*, the software must force the I2C_EN bit to 0 in the *Host Configuration* register and the PEC_EN bit to 0 in the *Host Control* register. The *Transmit Slave Address* register should be written with the slave device address in the Address field and the command code (0 or 1) in the RW bit.

Note

- The PEC_EN bit in the *Host Control* register must be written prior to the write operation in which the START bit is set in the same register.

3.6.2.2. Send/Receive Byte Command

For a *Send Byte* command, the *Transmit Slave Address* register must be written with the slave device address in the Address field and the direction of transfer (0) in the RW bit. The *Host Command* register should be written with the data byte to be sent.

For a *Receive Byte* command, the *Transmit Slave Address* register must be written with the slave device address in the Address field and the direction of transfer (1) in the RW bit. The data byte received will be stored in the *Host Data 0* register.

3.6.2.3. Write/Read Byte Command

For a *Write Byte* command, the *Transmit Slave Address* register must be written with the slave device address in the Address field and the direction (0) in the RW bit. The *Host Command* register should be written with the command code, and the *Host Data 0* register should be written with the data byte to be sent.

For a *Read Byte* command, the *Transmit Slave Address* register must be written with the slave device address in the Address field and the direction (1) in the RW bit. The *Host Command* register should be written with the command code. The data byte read will be stored in the *Host Data 0* register.

3.6.2.4. Write/Read Word Command

For a *Write Word* command, the *Transmit Slave Address* register must be written with the slave device address in the Address field and the direction (0) in the RW bit. The *Host Command* register should be written with the command code. The *Host Data 0* register should be written with the low data byte to be sent, and the *Host Data 1* register should be written with the high data byte to be sent.

For a *Read Word* command, the *Transmit Slave Address* register must be written with the slave device address in the Address field and the direction (1) in the RW bit. The *Host Command* register should be written with the command code. After command completion, the low data byte read will be stored in the *Host Data 0* register, and the high data byte read will be stored in the *Host Data 1* register.

3.6.2.5. Process Call Command

Before issuing a *Process Call* command, the software must force either the I2C_EN bit to 0 in the *Host Configuration* register or the PEC_EN bit to 0 in the *Host Control* register.

Note

- Executing a *Process Call* command with the I2C_EN and PEC_EN bits both set to 1 produces undefined results.

For executing this command, the *Transmit Slave Address* register must be written with the slave device address in the Address field and the direction (0) in the RW bit. The *Host Command* register should be written with the command code. The *Host Data 0* register should be written with the low data byte to be sent, and the *Host Data 1* register should be written with the high data byte to be sent. After command completion, the low data byte received will be stored in the *Host Data 0* register, and the high data byte received will be stored in the *Host Data 1* register.

Note

- When operating in I²C mode, with the I2C_EN bit in the *Host Configuration* register set, the implemented protocol changes in that the command code is not sent as part of the message.

3.6.2.6. Block Write/Read Command

Before issuing a *Block Write* command, the software must either force the I2C_EN bit to 0 in the *Host Configuration* register or force the PEC_EN bit in the *Host Control* register and the AAC bit in the *Auxiliary Control* register both to 0.

The *Block Write/Read* command may use the 32-byte buffer of the SMBus controller. This buffer can be enabled by setting the E32B bit of the *Auxiliary Control* register. For a write command, the software fills the 32-byte buffer with the data to be transmitted, and for a read command, the controller fills the buffer with the data received. The controller will only generate an interrupt after transmission or reception of 32 bytes, or when the byte count has been exhausted.

Note

- When operating in I²C mode, with the I2C_EN bit in the *Host Configuration* register set, the controller will not use the 32-byte buffer for any block command.

For a *Block Write* command, the *Transmit Slave Address* register must be written with the slave device address in the Address field and the direction (0) in the RW bit. The *Host Command* register should be written with the command code, and the *Host Data 0* register should be written with the byte count representing the number of bytes that will be sent. The byte count may not be 0. Depending on whether the 32-byte buffer has been enabled or not, the data bytes to be sent should be written to the controller's registers as specified next.

- If the 32-byte buffer has been enabled, the software will write up to 32 data bytes into the *Host Block Data Byte* register. The controller will send the slave device address, the command code, the byte count, and all the data bytes from the 32-byte buffer.
- If the 32-byte buffer has been disabled, the software will write a single data byte into the *Host Block Data Byte* register. After the controller sends the slave device address, the command code, and the byte count, it will send the data byte from the *Host Block Data Byte* register and will set the BYTE_DONE_STS bit in the *Host Status* register. If there are more bytes to send, the software will write the next data byte into the *Host Block Data Byte* register and will clear the BYTE_DONE_STS bit. The controller will then send the next data byte.

Note

- When operating in I²C mode, with the I2C_EN bit in the *Host Configuration* register set, the implemented *Block Write* protocol changes in that the byte count contained in the *Host Data 0* register is not sent as part of the message.

For a *Block Read* command, the *Transmit Slave Address* register must be written with the slave device address in the Address field and the direction (1) in the RW bit. The *Host Command* register should be written with the command code. After the controller sends the slave device address and the command code, it receives the byte count in the *Host Data 0* register. Depending on whether the 32-byte buffer has been enabled or not, the data bytes will be received differently, as specified next.

- If the 32-byte buffer has been enabled, the controller will store the received data bytes in the 32-byte buffer. If the byte count has been exhausted or the 32-byte buffer has been filled, the controller will generate an interrupt and set the BYTE_DONE_STS bit in the *Host Status* register. The software will then read the data bytes individually

from the *Host Block Data Byte* register, and then it will clear the BYTE_DONE_STS bit.

Note

- The software must perform a read of the *Host Control* register to reset the pointer into the 32-byte buffer prior to reading the *Host Block Data Byte* register.

- If the 32-byte buffer has been disabled, the controller will store a received byte in the *Host Block Data Byte* register. Then, the controller will generate an interrupt and set the BYTE_DONE_STS bit in the *Host Status* register. The software will read the data byte from the *Host Block Data Byte* register and will clear the BYTE_DONE_STS bit. The controller will then receive the next data byte.

3.6.2.7. I²C Read Command

The *I²C Read* command allows to perform a block read operation with certain I²C-compatible devices, such as serial EEPROM memories. For instance, this command allows access to devices that are using the I²C combined format cycle and require to send a single byte after the slave device address. Typically, this byte corresponds to an address (offset) within the memory chip.

When executing an *I²C Read* command, the SMBus controller sends the slave device address followed by the contents of the *Host Data 1* register. The controller then generates a repeated START condition, sends the slave device address again, and begins to receive a number of data bytes from the slave device. After the required number of data bytes have been received, the controller sends a NACK bit and generates a STOP condition, ending the transfer.

The *I²C Read* command only supports the 7-bit addressing mode of the I²C bus.

Before issuing an *I²C Read* command, the software must force both the PEC_EN bit in the *Host Control* register and the AAC bit in the *Auxiliary Control* register to 0.

Notes

- Executing an *I²C Read* command with the PEC_EN bit set to 1 produces undefined results.
- Execution of an *I²C Read* command is supported by the controller regardless of the setting of the I2C_EN bit in the *Host Configuration* register.

Before issuing an *I²C Read* command, the *Transmit Slave Address* register must be written with the slave device address in the Address field and the direction (0) in the RW bit. The *Host Data 1* register should be written with the byte to be sent to the slave device; for instance, this byte might be the starting address from which the contents of a memory chip should be read.

Notes

- For the *I²C Read* command, the RW bit in the *Transmit Slave Address* register must be set to 0; this setting is different from the setting for other read commands, for which the RW bit must be set to 1, corresponding to the read direction.
- Since a single byte is sent after the slave device address, the *I²C Read* command cannot be used with devices that require to send more than one byte after the slave device address, such as with memories that require a two-byte address.

For receiving the data from the slave device, the software should perform the following operations for each data byte:

1. Check the status of the BYTE_DONE_STS bit in the *Host Status* register and wait until the controller sets this bit to 1, which means that it has received a data byte and stored it in the *Host Block Data Byte* register.

2. Read the contents of the *Host Block Data Byte* register into a buffer in memory.
3. Clear the BYTE_DONE_STS bit in the *Host Status* register.
4. If the byte received is the next to last byte, set the LAST_BYTE bit in the *Host Control* register and continue with Step 1 (setting this bit will cause the controller to send a NACK bit instead of an ACK bit after receiving the last byte). Otherwise (if the byte received is not the next to last byte), continue with Step 1.
5. If the byte received is the last byte, clear the LAST_BYTE bit in the *Host Control* register and the operation is completed. Otherwise, continue with Step 1.

3.6.2.8. Block Write-Block Read Process Call Command

The *Block Write-Block Read Process Call* command implements the SMBus protocol with the same name, which has been described in Section 3.4.8. However, the following differences exist between this command and the corresponding protocol described in the SMBus specification:

- In the command implementation, none of the byte counts, write byte count (M) and read byte count (N), can be zero, while in the protocol specification any byte count can be zero.
- In the command implementation, the combined byte count (M + N) must not exceed 32, while in the protocol specification the combined byte count must not exceed 255.

Note

- The E32B bit in the *Auxiliary Control* register must be set before issuing this command.

For executing the *Block Write-Block Read Process Call* command, the *Transmit Slave Address* register must be written with the slave device address in the Address field and the direction (0) in the RW bit. The *Host Command* register should be written with the command code. The *Host Data 0* register should be written with the write byte count indicating the number of data bytes that will be sent. The data bytes to be sent should be written into the *Host Block Data Byte* register, byte by byte; they will be stored in the 32-byte buffer.

After sending the slave device address, command code, write byte count, and data bytes from the 32-byte buffer, the controller receives the read byte count and stores it in the *Host Data 0* register. The controller then receives the number of data bytes indicated by the read byte count, stores them in the 32-byte buffer, generates an interrupt, and sets the BYTE_DONE_STS bit in the *Host Status* register. The software should perform a read of the *Host Control* register to reset the pointer into the 32-byte buffer, should read the data bytes individually from the *Host Block Data Byte* register, and clear the BYTE_DONE_STS bit.

3.7. Using the Intel SMBus Controller with I²C Devices

The I²C bus is used by a wide variety of devices, such as EEPROM memories, real-time clocks, and various types of sensors, including temperature sensors. The Intel SMBus controller can communicate with many of these devices in addition to be able to communicate with native SMBus devices.

There are several differences between the I²C bus cycles and the SMBus protocols; some of these differences have been discussed in Section 3.5. However, by a careful selection of specific controller commands and by certain register settings, it is often possible to establish a communication between the SMBus controller and I²C slave devices. For instance, the SMBus controller includes a setting to enable the I²C mode; this is achieved by setting the I2C_EN bit in the *Host Configuration* register. In addition, the controller provides a specific I²C command called *I²C Read*, which has been described in Section 3.6.2.7. Nonetheless, neither of these settings places the controller into a 100% I²C mode.

3.7.1. I²C Bus Cycles

The I²C bus specification defines three basic bus cycle types: write cycle, read cycle, and combined format cycle type. Each cycle type operates in block mode, being able to transfer more than one data byte. In this section, the basic I²C bus cycle types are presented, and the differences between each cycle type and a corresponding SMBus protocol are underlined.

3.7.1.1. I²C Write Cycle

The I²C write cycle is illustrated in Figure 3.13. The master device can send multiple data bytes to the slave device; the number of bytes is not limited. The number of bytes transferred is determined by the slave device; after a certain number of bytes received, the slave device sends a NACK bit, and then the master device generates a STOP condition. This cycle differs from the SMBus *Block Write* protocol, in which the master device determines the transfer size by only sending the required number of bytes during the transfer.

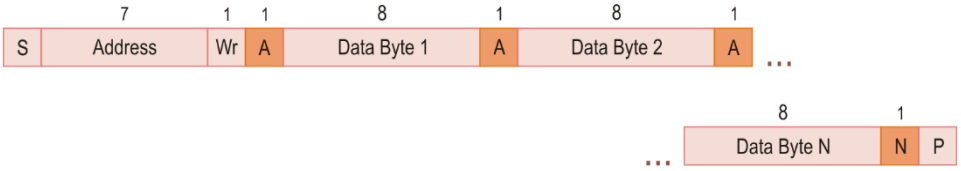

Figure 3.13. I²C write cycle.

Another difference from the SMBus *Block Write* protocol is that during the I²C write cycle the data bytes are sent immediately after the slave device address, while during the SMBus protocol a command code and a byte count are sent before sending the data bytes. This may prevent the *Block Write* protocol from being used with I²C devices. Another potential problem is that an I²C slave device will send a NACK bit after receiving the last data byte, while, according to the SMBus protocol, the master device expects an ACK bit from the slave device even after the last data byte.

3.7.1.2. I²C Read Cycle

The I²C read cycle is illustrated in Figure 3.14. The master device can read multiple data bytes from the slave device; the number of bytes is not limited. The number of bytes transferred is determined by the master device; after a certain number of data bytes received, the master device sends a NACK bit, and then it generates a STOP condition.

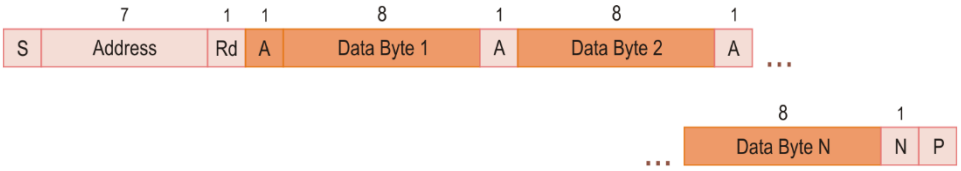

Figure 3.14. I²C read cycle.

The I²C read cycle differs significantly from the SMBus *Block Read* protocol. During the I²C read cycle, the data bytes are received by the master device immediately after it sends the slave device address. During the SMBus *Block Read* protocol, after sending the slave device address, the master device sends a command code, then it generates a repeated START condition, sends the slave device address again, and expects a read byte count from the slave device before the data bytes themselves. Due to these differences, the SMBus *Block Read* protocol cannot be used to communicate with devices that use the I²C read cycle.

3.7.1.3. I²C Combined Format Cycle

The I²C combined format cycle is illustrated in Figure 3.15. In the first part of this cycle type, the master device sends the slave device address, after which it sends a number of data bytes to the slave device until the slave device sends a NACK bit. In the second part of this cycle type, the master device generates a repeated START condition, sends the slave device address again, receives a certain number of data bytes from the slave device, sends a NACK bit, and generates a STOP condition.

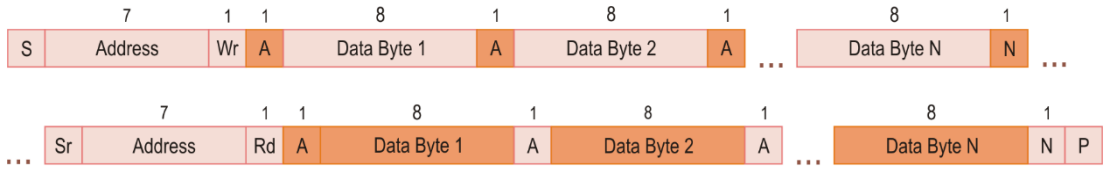

Figure 3.15. I²C combined format cycle.

For communication with I²C devices that use the I²C combined format cycle, a possibility would be to use the SMBus *Block Write-Block Read Process Call* protocol. However, one problem is that during this protocol the master device sends a command code and a write byte count, which are not part of the I²C combined format cycle. Another problem is that in the SMBus protocol, after generating a repeated START condition and sending the slave device address, the master device expects to receive a read byte count, which is also not part of the I²C combined format cycle. Therefore, the SMBus *Block Write-Block Read Process Call* protocol cannot be used to communicate with devices that use the I²C combined format cycle.

3.7.2. I²C-Compatible SMBus Controller Commands

The protocol used by several commands of the SMBus controller can be changed to a certain extent in order to be more compatible with the I²C bus cycles by setting the I2C_EN bit in the *Host Configuration* register. The effects of setting the I2C_EN bit are the following:

- The *Block Write* command will not send the byte count;
- The *Process Call* command will not send the command code.

Note

- Setting the I2C_EN bit and selecting the *I²C Read* command should not be confused. Setting the I2C_EN bit modifies the execution of the *Block Write* and *Process Call* commands, while selecting the *I²C Read* command allows to implement a particular I²C combined format cycle, as described in Section 3.6.2.7.

When the SMBus controller must be used with a certain I²C device, there is no general rule to determine if communication with that device is possible. The device's datasheet should be analyzed carefully to determine which I²C cycles are needed and whether a particular cycle can be created by the SMBus controller. The following commands of the SMBus controller can be considered compatible with certain I²C devices under the specified conditions:

- *Quick Command*: Sends the slave device address and the R/W# bit to the device; this bit specifies a command for the device.
- *Send Byte*: Sends a single data byte from the *Host Command* register to the device.
- *Receive Byte*: Receives a single data byte from the device into the *Host Data 0* register.
- *Write Byte*: Sends two data bytes to the device, the first from the *Host Command* register and the second from the *Host Data 0* register.

- *Read Byte*: Sends a single data byte from the *Host Command* register to the device and receives a single data byte from the device into the *Host Data 0* register.
- *Write Word*: Sends three data bytes to the device, the first from the *Host Command* register, the second from the *Host Data 0* register, and the third from the *Host Data 1* register.
- *Read Word*: Sends a single data byte from the *Host Command* register to the device and receives two data bytes from the device, the first into the *Host Data 0* register and the second into the *Host Data 1* register.
- *Process Call* with I2C_EN = 1: Sends two data bytes to the device, the first from the *Host Command* register and the second from the *Host Data 0* register, and receives two data bytes from the device, the first into the *Host Data 0* register and the second into the *Host Data 1* register.
- *Block Write* with I2C_EN = 1: Sends *N* data bytes to the device, the first byte from the *Host Command* register, and the remaining *N*–1 bytes from the 32-byte buffer or individually from the *Host Block Data Byte* register.

Note

- The *Block Write* command of the SMBus controller can transfer a maximum of 32 data bytes when the 32-byte buffer is enabled, although the I^2C bus does not limit the transfer size.

- *I^2C Read*: Sends a single data byte from the *Host Data 1* register to the device and receives *N* data bytes from the device, individually into the *Host Block Data Byte* register.

3.7.3. Examples for Using I^2C Devices with the SMBus Controller

In this section, two examples are presented for connecting I^2C-compatible devices to the SMBus controller. The first example is for connecting a serial EEPROM device, and the second example is for connecting an analog-to-digital converter.

3.7.3.1. Using the Microchip 24LC01B EEPROM

The Microchip Technology 24LC01B device is a serial EEPROM with a capacity of 1 Kbit. The device is organized as one block of 128 x 8-bit memory. This device is commonly used in DIMM memory modules as the SPD (*Serial Presence Detect*) EEPROM.

We will only refer to the read operations supported by this memory, although the device also supports write operations. This memory supports the following read operations: current address read; random read; sequential read.

Current Address Read

The 24LC01B device has an internal address pointer to the last byte accessed during a previous operation. This pointer is initialized to zero when the device is powered up and is incremented automatically after each read or write access. The current address read cycle is illustrated in Figure 3.16.

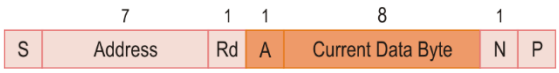

Figure 3.16. Current address read cycle of the Microchip 24LC01B memory device.

For the current address read cycle, the controller sends the slave device address and the R/W# bit set to 1 (Rd bit). The device sends an ACK bit and then sends the current data

byte from the memory. It is easy to observe that this cycle can be created using the *Receive Byte* command of the SMBus controller.

Random Read

A random read operation allows to access any memory location regardless of previous accesses. The random read cycle is illustrated in Figure 3.17. After sending the slave device address and the R/W# bit set to 0 (Wr bit), the controller sends the 8-bit address of the location to be accessed, then it generates a repeated START condition and sends the slave device address again. The device sets the internal address pointer and then sends the data byte from the addressed location.

Figure 3.17. Random read cycle of the Microchip 24LC01B memory device.

Reviewing the SMBus protocols, it can be seen that the random read cycle is similar to the SMBus *Read Byte* protocol. The *Read Byte* command of the SMBus controller can be used to create this cycle, with the *Host Command* register containing the location address.

Sequential Read

A sequential read operation can be initiated by the controller similarly to a random read operation, by sending the slave device address, the Wr bit, the address of the first location to be accessed, followed by a repeated START condition and the slave device address (Figure 3.18). The device sets the internal address pointer and then sends a block of data bytes starting from the addressed location. The device is expecting the controller to terminate the cycle by sending a NACK bit after the last data byte (n + x) and generating a STOP condition.

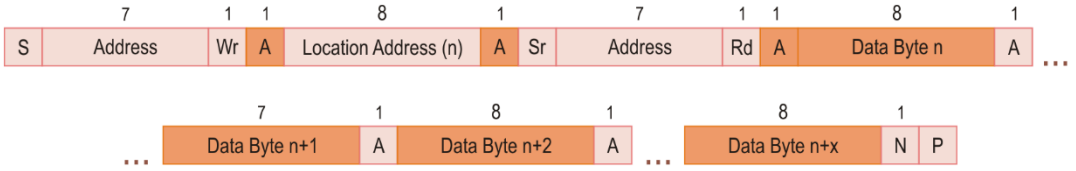

Figure 3.18. Sequential read cycle of the Microchip 24LC01B memory device.

This cycle can be created using the *I²C Read* command of the SMBus controller. The *Host Data 1* register must be loaded with the location address before issuing the command. Software should monitor the data bytes read and set the LAST_BYTE bit in the *Host Control* register after receiving the next to last byte to ensure that the controller sends a NACK bit after the last byte.

3.7.3.2. Using the LTC2481 Analog-to-Digital Converter

The LTC2481 device is a 16-bit analog-to-digital converter by Linear Technology. The device can be configured through an I²C interface to provide a programmable gain from 1 to 256, to digitize an external signal or an integrated sensor temperature, select line frequency noise rejection (50 Hz, 60 Hz, or simultaneous 50Hz and 60 Hz), and select a 2x speed up mode.

The LTC2481 converter has two registers, an output register, and a configuration register. The output register contains the last conversion result. The configuration register is user-programmable and allows to set the converter operation mode. The device supports several write and read operations. Write operations allow to initiate a new conversion and to set the operation mode of the converter. Read operations allow to read up to three bytes from the device, which include the last conversion result and the contents of the configuration register.

Initiating a New Conversion

A new conversion may be initiated with a simple write cycle. For this cycle, the SMBus controller sends the device address and a Wr bit (Figure 3.19). The device sends an ACK bit to acknowledge the write cycle. When the controller finishes the cycle by generating a STOP condition, the device initiates a new conversion. The result of this conversion can be retrieved with a subsequent read cycle.

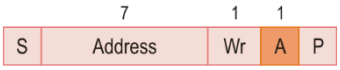

Figure 3.19. Write cycle for initiating a new conversion for the LTC2481 device.

This write cycle can be created using the *Quick Command* of the SMBus controller.

Initiating a New Conversion with Configuration Updating

A modified write cycle can be used to initiate a new conversion and to write a new configuration into the configuration register. After the SMBus controller sends the device address and a Wr bit, the device acknowledges the write cycle. The controller then sends a data byte containing the new configuration (Figure 3.20). The device sends an ACK bit, and the controller generates a STOP condition, when the device initiates a new conversion.

Figure 3.20. Write cycle for initiating a new conversion and updating the configuration of the LTC2481 device.

This modified write cycle can be created using the *Send Byte* command of the SMBus controller.

Continuous Read

When the configuration does not need to change after each conversion cycle, the conversion result can be read continuously. When the device finishes a conversion, it may be addressed for a read operation. At the end of a read operation, a new conversion begins. If the conversion cycle is not finished and the device receives a valid command, it sends a NACK bit indicating that the conversion cycle is in progress.

The cycle for reading a 24-bit value from the converter is illustrated in Figure 3.21. After receiving the slave device address and a Rd bit, the converter sends three data bytes to the controller, which then sends a NACK bit and generates a STOP condition.

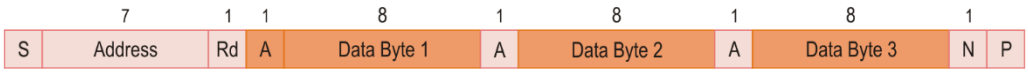

Figure 3.21. Read cycle for reading the last conversion result and the configuration register of the LTC2481 device.

Since this cycle requires three bytes to be read from the device, it will have to be created using one of the block commands of the SMBus controller. However, none of the available commands allows to create this cycle, and therefore the continuous read operation cannot be issued using the SMBus controller.

Continuous Write/Read

After a conversion cycle is completed, the configuration register of the LTC2481 converter can be updated and the three data bytes can be read from the device with the cycle illustrated in Figure 3.22.

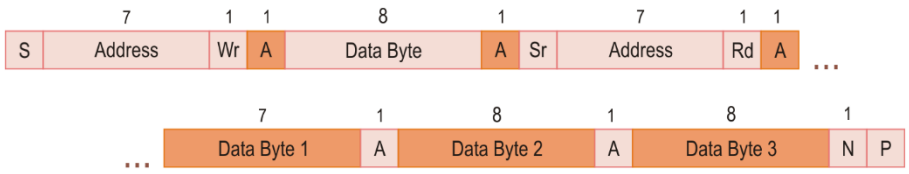

Figure 3.22. Write, read, start conversion cycle of the LTC2481 device.

For this cycle, the controller sends the slave device address, a Wr bit, and a data byte containing the configuration information. The controller then generates a repeated START condition and sends the slave device address again. The device returns three data bytes to the controller, which sends a NACK bit and generates a STOP condition.

This cycle can be created using the *I²C Read* command of the SMBus controller. The *Host Data 1* register must be loaded with the configuration information before issuing the command. The software should set the LAST_BYTE bit in the *Host Control* register after receiving the second data byte from the device, which will cause the controller to send a NACK bit and to generate a STOP condition after the third byte received.

3.8. Applications

3.8.1. Answer the following questions:

a. What are the improvements of SMBus introduced in version 2.0 and version 3.0 of the SMBus specification?

b. What are the main differences between SMBus and the I²C bus?

c. What is the difference between the I²C read cycle and the SMBus *Block Read* protocol?

d. What is the effect of setting the I2C_EN bit in the *Host Configuration* register of the Intel SMBus controller compared to selecting the *I²C Read* command of this controller?

3.8.2. Open the project created for Application 2.7.5 of the *PCI Express Bus* laboratory. Copy to the project folder the SMBus-e.h file, available on the laboratory web page in the SMBus-e.zip archive. Add to the project the SMBus-e.h header file, open the AppScroll-e.cpp source file, and add a `#include` directive to include the SMBus-e.h header file. In the AppScroll-e.cpp source file, write a function that returns the base address (of type `WORD`) for the I/O registers mapped in the I/O space of the SMBus controller. The input parameter of this function is the pointer to the PCIe configuration header of the SMBus controller, determined in Application 2.7.5. Call this function, store the base address returned by the function in a global variable, and display the base address.

Notes

- In the PCIe configuration header, the register containing the base address for the I/O registers mapped in the I/O space of the SMBus controller is SMB_BASE. The base address is in the low word of this 32-bit register. Bit 0 of the base address read from this register should be cleared to 0.
- For an AMD processor, if the SMBus controller is found on bus 0, device 20, and function 0, the base address of the I/O registers cannot be determined by reading the registers of the PCIe configuration header. In this case, the function will return a base address of 0x0B00.

3.8.3. Extend Application 3.8.2 by writing a function that aborts the current transaction of the SMBus controller. The input parameter of the function is the base address for the I/O registers mapped in the I/O space of the SMBus controller. The function sets the KILL bit in the *Host Control* register, waits for a certain time (e.g., 100 ms), and then clears the KILL

bit. The function returns 0 if the current transaction has been aborted and 1 otherwise. The transaction has been aborted if the HOST_BUSY bit in the *Host Status* register is not set and the FAILED bit in the same register is set.

3.8.4. Continue Application 3.8.2 by writing a function that sends the *Receive Byte* command to an SMBus device. The input parameters of the function are the handler to the application window (of type `hWnd`), and the SMBus device address (of type `BYTE`). The function returns 0 if the *Receive Byte* command completed successfully, in which case the INTR bit in the *Host Status* register is set. The function returns 1 if the command execution is completed with an error, when one of the following bits of the *Host Status* register is set: FAILED, BUS_ERR, or DEV_ERR.

Notes

- Define the offsets of the registers used in the function and the bitmasks needed for these registers with `#define` directives at the beginning of the AppScroll-e.cpp file.
- When calling the HW driver functions to access a register, the address of the register is formed by adding its offset to the base address of the I/O registers of the SMBus controller.

3.8.5. Continue Application 3.8.2 to identify the devices connected to the SMBus of the computer. In the main function (`AppScroll`) of this application, after determining the base address for the I/O registers of the SMBus controller with the function written for Application 3.8.2, send the *Receive Byte* command to devices with addresses between 0x10 and 0x7F with the function written for Application 3.8.4. If the execution of this command is completed successfully, display a message indicating that an SMBus device has been found and display the device address. Additionally, display the type of the device detected, based on the following address ranges used by some SMBus devices:

- 0x18..0x1F: Thermal sensors of an SPD memory;
- 0x30..0x37: Write protection for an SPD memory;
- 0x40..0x47: Real-time clock;
- 0x50..0x57: SPD memory.

3.8.6. Extend the application that identifies the devices connected to the SMBus by writing a function that sends the *Read Byte* command to an SMBus device. The input parameters of the function are the handler to the application window (of type `hWnd`), the SMBus device address (of type `BYTE`), and the command code (of type `BYTE`). The function returns the same values as the function that sends the *Receive Byte* command, written for Application 3.8.4.

3.8.7. Extend the application that identifies the devices connected to the SMBus with a function that reads and displays the contents of an SPD memory. The input parameters of the function are the handler to the application window (of type `hWnd`) and the SMBus device address (of type `BYTE`). The function returns the same values as the function that sends the *Receive Byte* command, written for Application 3.8.4. The operations that should be performed by this function are the following:

1. Send the *Read Byte* command to the device (an SPD memory), with the command code set to 0. When receives this command, the SPD memory will reset its internal pointer and will return the first byte from the memory.
2. Store the received byte in the first location of a 512-bytes buffer, and based on this byte, determine the number of bytes used in the SPD memory. This number is indicated by bits 3..0 of the byte received. When these bits are 0001, 128 bytes are used in the SPD memory, when they are 0010, 256 bytes are used, and when they are 0011, 384 bytes are used. For other combinations, it is assumed that the number of bytes used is 512.

3. Send repeatedly (in a loop) the *Receive Byte* command to the device and store the bytes received starting with the second location of the 512-byte buffer. The iteration count should be the number of bytes used in the SPD memory minus 1.
4. Display the contents of the SPD memory. In each line, display three digits in decimal representing the offset of an 8-byte area in the memory, followed by 8 data bytes in hexadecimal.

In the main function of the application, call the function described previously for each SPD memory detected on the SMBus; SPD memories have addresses between 0x50 and 0x57.

3.8.8. Modify Application 3.8.7 to decode part of the information read from an SPD memory rather than displaying the memory contents. To decode the contents of the memory, use the SPD-e.h header file, available on the laboratory web page in the SPD-e.zip archive. The following information should be decoded:

- SPD revision;
- DRAM device type;
- Module type;
- SDRAM density (capacity);
- Number of internal banks;
- Module nominal voltage;
- Memory type and bus frequency, based on the minimum cycle time;
- Minimum CAS latency time (ns);
- Minimum RAS to CAS delay time (ns);
- Minimum RAS precharge time (ns).
- Module manufacturer;
- Module serial number;
- Module part number;
- DRAM manufacturer.

Note

- The SPD-e.h header file defines structures to simplify decoding the SPD memory contents. Each structure contains a byte and a pointer to a character string. Based on the value of a byte in the SPD memory, it is possible to directly display the character string corresponding to the byte meaning. The file also includes comments describing the meaning of the most important bytes of the SPD memory.

3.8.9. Extend Application 3.8.7 by writing a function that sends the *I²C Read* command to an SMBus device and stores in a buffer the specified number of bytes received from the device. The input parameters of the function are the following: the handler to the application window (of type `hWnd`); the SMBus device address (of type `BYTE`); the command code (of type `BYTE`); a pointer to the receive buffer (of type `PBYTE`); and the number of bytes to be received (of type `int`). If the specified number of bytes to be received is 0, the function determines the number of bytes to be received as described in Step 2 of the operations to be performed for Application 3.8.7. The function returns the same values as the function that sends the *Receive Byte* command, written for Application 3.8.4. The function should perform the operations described in Section 3.6.2.7 for receiving each data byte from the device.

3.8.10. Modify Application 3.8.7 in order to use the *I²C Read* command for reading the contents of an SPD memory instead of the *Read Byte* and *Receive Byte* commands. Change the function that reads and displays the contents of an SPD memory so that it calls the function written for Application 3.8.9 for sending the *I²C Read* command to the SPD memory. When calling this function, specify the value 0 for the command code (which will reset the internal pointer of the memory) and the value 0 for the number of bytes to be received (therefore, the number of bytes to be received will be determined from the first byte read from the memory).

Bibliography

[1] Fan, Roger, “SMBus Quick Start Guide”, Application Note AN4471, Rev. 1, Freescale Semiconductor Inc., 2012, http://cache.freescale.com/files/32bit/doc/app_note/AN4471.pdf.

[2] Fleming, Sam, “Interfacing I²C Devices to an Intel SMBus Controller”, White Paper, Intel Corporation, 2009, http://www.intel.com/content/dam/www/public/us/en/documents/white-papers/smbus-controller-i2c-devices-paper.pdf.

[3] Intel Corporation, “Intel 8 Series/C220 Series Chipset Family Platform Controller Hub (PCH)”, Datasheet, May 2014, http://www.intel.com/content/dam/www/public/us/en/documents/datasheets/8-series-chipset-pch-datasheet.pdf.

[4] JEDEC Solid State Technology Association, "SPD Annex L: Serial Presence Detect (SPD) for DDR4 SDRAM Modules", Release 6, SPD4.1.2.L-6, Nov. 2020.

[5] JEDEC Solid State Technology Association, “Standard Manufacturer’s Identification Code”, JEP106BC, Feb. 2021.

[6] Linear Technology Corporation, “LTC2481 – 16-Bit ΔΣ ADC with Easy Drive Input Current Cancellation and I²C Interface”, 2005, http://cds.linear.com/docs/en/datasheet/2481fd.pdf.

[7] Melexis Microelectronic Integrated Systems, “SMBus Communication with MLX90614”, Application Note, Rev. 004, 2008, http://www.generationrobots.com/media/SMBus-communication-with-MLX90614.pdf.

[8] Microchip Technology Inc., “24AA01/24LC01B 1K I²C Serial EEPROM”, Datasheet DS21711J, 2009, http://ww1.microchip.com/downloads/en/DeviceDoc/21711J.pdf.

[9] System Management Interface Forum, “System Management Bus (SMBus) Specification”, Version 3.0, 20 December 2014, http://pmbus.org/Assets/PDFS/Public/SMBus_3_0_20141220.pdf.